**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**



**Simulator de cozi la magazin. Documentație**

Grigorescu M. Alexandru

Departamentul de calculatoare, Universitatea Tehnică, 30223

Tehnici de programare

Prof. dr. ing. Tudor Cioara

21 Aprilie, 2024

# Cuprins

[Obiectivul temei 3](#_Toc161591649)

[Determinarea unor cazuri de utilizare 3](#_Toc161591650)

[Proiectarea algoritmilor de prelucrare a polinoamelor 3](#_Toc161591651)

[Modelarea tipurilor de date utilizate 3](#_Toc161591652)

[Crearea unei arhitecturi flexibile 3](#_Toc161591653)

[Implementarea unei interfețe grafice prietenoase cu utilizatorul 4](#_Toc161591654)

[Analiza problemei, modelare, scenarii, cazuri de utilizare 5](#_Toc161591655)

[Cerințe Funcționale 5](#_Toc161591656)

[Cerințe non funcționale 5](#_Toc161591657)

[Cazuri de utilizare 5](#_Toc161591658)

[Proiectarea algoritmilor de prelucrare a polinoamelor 7](#_Toc161591659)

[Modelarea structurilor de date 7](#_Toc161591660)

[Proiectare 9](#_Toc161591661)

[Routing 9](#_Toc161591662)

[Interfața 9](#_Toc161591663)

[Integrarea interfeței 10](#_Toc161591664)

[Structura pachetelor 11](#_Toc161591665)

[Containere de injecție de dependințe 12](#_Toc161591666)

[Routing 13](#_Toc161591667)

[Procesarea operațiilor 13](#_Toc161591668)

[Testare 14](#_Toc161591669)

[Scenarii 15](#_Toc161591670)

[Concluzii 16](#_Toc161591671)

[Bibliografie 17](#_Toc161591672)

# Obiectivul temei

Este dorită elaborarea unui sistem de administrarea al cozilor de la magazine cu vizualizare live. Scopul sistemului este de a organiza cozile pentru a obține timpul optim de așteptare pentru clienții magazinului.

### Determinarea unor cazuri de utilizare

Cazurile de utilizare (sau usecase-urile) sunt scenarii specifice în care utilizatorii vor interacționa cu sistemul nostru și vor utiliza funcționalitățile oferite de acesta. Prin definirea și înțelegerea acestor usecase-uri, putem ghida procesul de dezvoltare și ne asigurăm că aplicația noastră va satisface nevoile utilizatorilor într-un mod complet și eficient. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 2, unde subiectul este explorat în profunzime.

### Proiectarea algoritmilor de prelucrare a polinoamelor

Procesul de proiectare a algoritmilor pentru manipularea cozilor constituie prima etapă în dezvoltarea aplicației propuse.

### Modelarea tipurilor de date utilizate

Alegerea unor structuri de date adecvate este crucială pentru eficiența algoritmilor implementați. De asemenea, este important să se țină cont de aspectele legate de gestionarea memoriei și de eficiența în timpul de execuție. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 2, unde subiectul este explorat în profunzime.

### Crearea unei arhitecturi flexibile

O arhitectură bine definită și flexibilă reprezintă o condiție prealabilă pentru o dezvoltare rapidă și eficientă a aplicației. Prin adoptarea unor principii de proiectare software, cum ar fi modularitatea, coeziunea și cuplajul redus, se asigură o structură a codului care poate fi ușor extinsă și modificată în viitor, în funcție de cerințele și feedback-ul utilizatorilor. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 3, unde subiectul este explorat în profunzime.

### Implementarea unei interfețe grafice prietenoase cu utilizatorul

Interfața grafică reprezintă componenta prin care utilizatorii interacționează cu aplicația și, în consecință, joacă un rol crucial în experiența lor de utilizare. Crearea unei interfețe grafice intuitive și prietenoase, care să faciliteze utilizatorilor accesul și utilizarea funcționalităților oferite de aplicație, necesită o atenție deosebită acordată designului ergonomic, navigației intuitive și feedback-ului vizual. Informații suplimentare și detaliate sunt furnizate în cadrul Capitolului 3, unde subiectul este explorat în profunzime.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

### Cerințe Funcționale

Aplicația *Marketplace Simulator x2000* trebuie să ofere următoarele servicii:

* Trebuie să permită utilizatorului să seteze simularea
* Trebuie să permită utilizatorului să poată manipula simularea asemenea unui video (opțiuni de ‘play’, ‘pause’, ‘next’, ‘previous’)
* Trebuie să arate statusul simulării în timp real
* Trebuie să permită utilizatorului să vadă log-urile în orice moment al simulării
* Trebuie să ofere KPI-uri la finalul simulării (timp mediu de așteptare)

### Cerințe non funcționale

Aplicația *Marketplace Simulator x2000* trebuie să aibă următorul comportament:

* Trebuie să informeze utilizatorul în cazul erorilor de sistem
* Trebuie să aibă o interfață prietenoasă cu utilizatorul
* Trebuie să funcționeze pe orice tip de calculator/laptop
* Trebuie să fie construit modular, astfel încât adăugarea noilor funcționalități să fie rapidă

### 

### Cazuri de utilizare

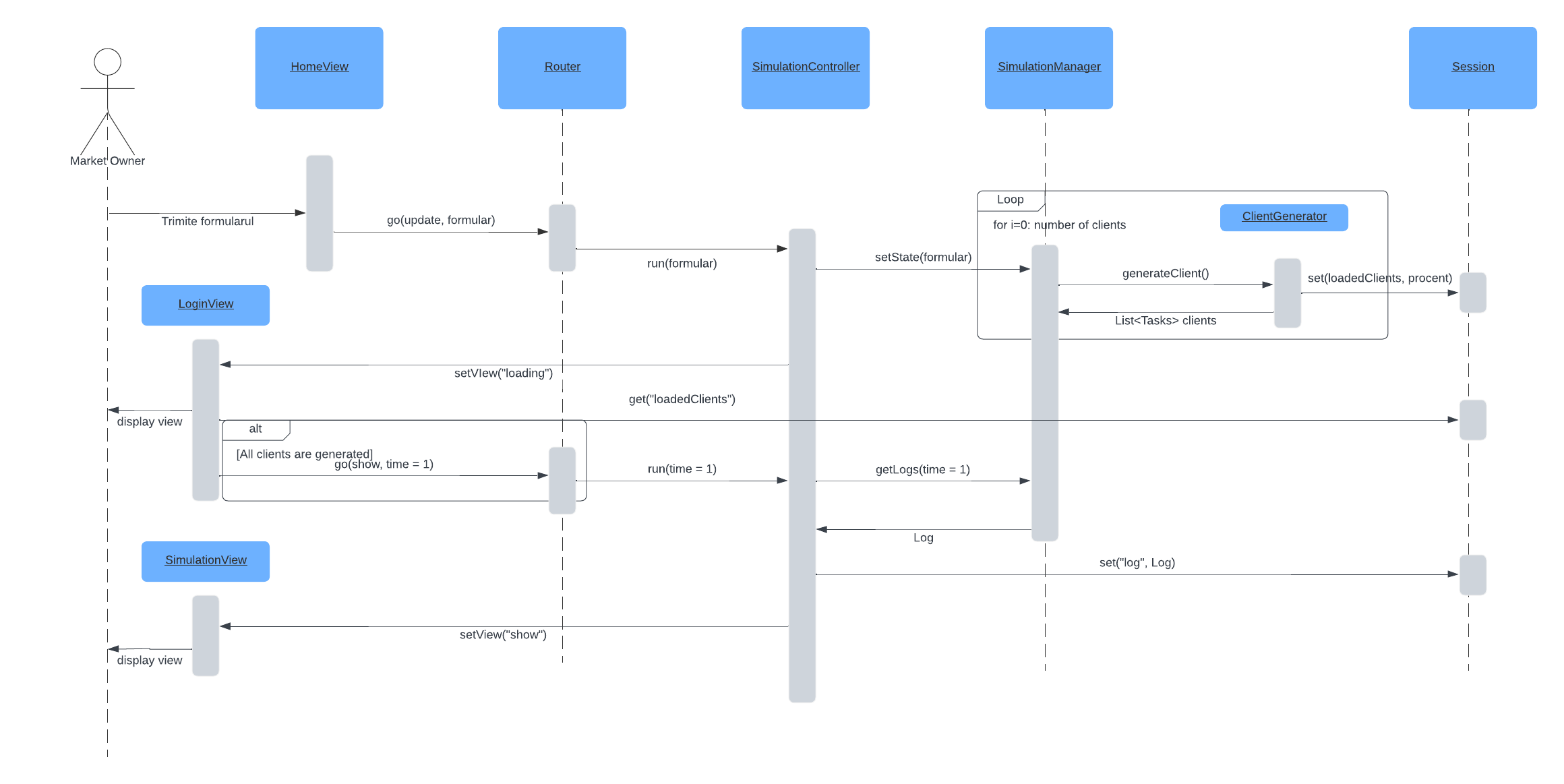
Diagrama alăturată prezintă toate cazurile de utilizare tratate de *Marketplace Simulator x2000.* În continuare se poate observa cum fiecare request al utilizatorului este procesat de sistem.

**Caz de utilizare:** Inițializarea simulării

**Scenariu de succes**:

1. Este introdus număr de clienți ce vor fi simulați
2. Este introdus număr de case de marcat
3. Este introdusă durata simulării
4. Sunt introduse timpurile minim și maxim în care clienți se pot pune la casă
5. Sunt introduse timpurile minim și maxim de scanare al coșului și plată.
6. Este apăsat butonul ‘Start simulation’
7. Aplicația încarcă pagina simulării

**Secvențe alternative:**

1. Sunt introduse numere mai mici sau egale cu 0 în oricare din field-uri
   1. Apare eroare pe ecran și field-urile sunt setate la valorile trimise de utilizator
   2. Așteptă remedierea problemei de către utilizator
2. Timpul minim este mai mic decât cel maxim (atât pentru sosire cât și pentru scanarea coșului)
   1. Apare eroare pe ecran și field-urile sunt setate la valorile trimise de utilizator
   2. ****Așteptă remedierea problemei de către utilizator

**Caz de utilizare:** Folosirea funcționalității de ‘step’

**Scenariu de succes**:

1. Simularea este inițializată
2. Este apăsat butonul ‘step’
3. Simularea merge înainte cu un pas (merge la timpul t+1)

**Caz de utilizare:** Folosirea funcționalității de ‘back’

**Scenariu de succes**:

1. Simularea este inițializată
2. Este apăsat butonul ‘back’
3. Simularea merge în urmă cu un pas (merge la timpul t-1)

**Caz de utilizare:** Folosirea funcționalității de ‘play’

**Scenariu de succes**:

1. Simularea este inițializată
2. Este apăsat butonul ‘play’
3. Simularea merge automat în fața cu o viteză de 300ms pe cadru
4. La finalul simulării apare un ecran cu toate log-urile generate de simulare

**Caz de utilizare:** Folosirea funcționalității de ‘pause’

**Scenariu de succes**:

1. Simularea este inițializată
2. Este apăsat butonul ‘star’
3. Simularea merge automat în fața cu o viteză de 300ms pe cadru
4. Este apăsat butonul ‘pause’
5. Simularea se oprește la timpul curent

# A diagram of a computer system Description automatically generatedProiectare

Întrucât aplicația dispune de o interfață grafică am ales o arhitectură de tipul MVC. Modul în care datele sunt transmise de la utilizator la aplicație și înapoi la utilizator este descris în diagramă de mai jos (figura 1). Totodată datorită volumului de muncă mare asincron am ales să combin arhitectura MVC cu una bazată pe evenimente. În următoarele secțiuni voi detalia modul de funcționare al sistemului.

### Routing

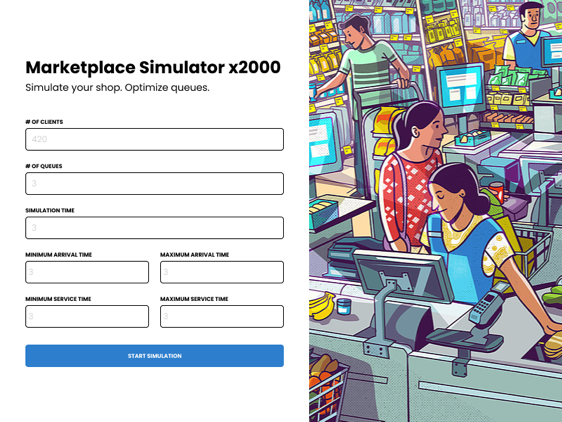
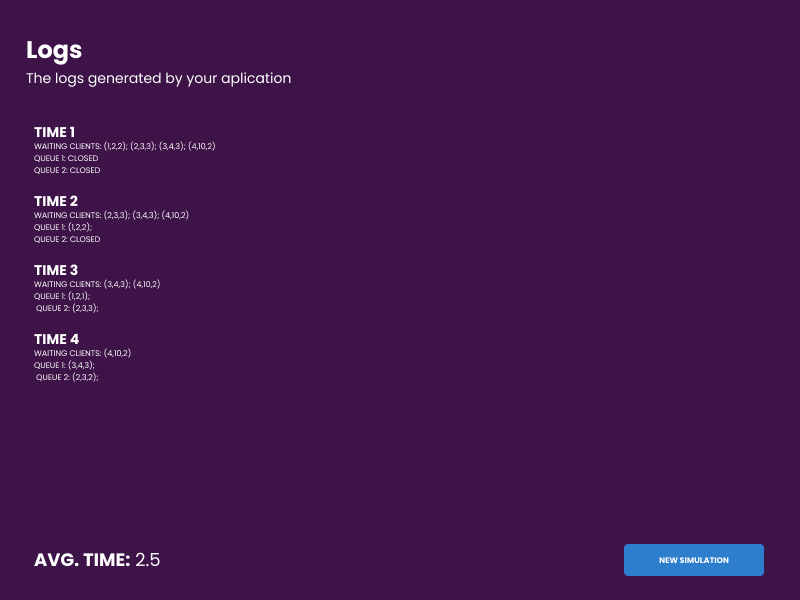
A diagram with text and a black arrow

Description automatically generated with medium confidencePentru a asigura o dezvoltare rapidă pe viitor, am creat un sistem de rutare care permite adăugarea rapidă și ușoară a rutelor noi. În cadrul aplicației, există o singură instanță a Router-ului care poate fi accesată din orice loc. Procesul începe odată ce cererea este trimisă către Router, care o direcționează către ruta corespunzătoare. Apoi, cererea este trecută către o clasă numită ControllerRunner, care, la rândul său, apelează metoda action din controler, având cererea ca parametru. Detaliile acestei implementări sunt prezentate în figura 3. Prin acest sistem, se îmbunătățește flexibilitatea și eficiența în gestionarea rutelor și a controlerelor, permițând o dezvoltare agilă și o adaptare rapidă la cerințele viitoare.

### Evenimente

das

### Interfața grafică

Interfața a fost proiectată folosind Figma, o unealtă ce permite crearea rapidă și ușoară a design-urilor. Aplicația poate fi în una dintre cele trei stări: inițială, în curs de completare, sau în stare de succes sau eroare. Abordarea în crearea design-ului a fost minimalistă, evitând supraîncărcarea utilizatorului cu toate opțiunile disponibile încă de la început. În schimb, sunt prezentate doar opțiunile relevante pentru stadiul curent al aplicației. Pentru a îmbunătăți experiența utilizatorului, am inclus și o schemă de culori pentru text, care reprezintă polinomul în curs de scriere. La finalizarea procesului, utilizatorul își selectează operația dorită prin intermediul unui meniu drop-down. Orice rezultate sau erori generate sunt afișate la finalul secțiunii de text. Aceste elemente contribuie la o experiență mai plăcută și mai intuitivă pentru utilizator.

### Integrarea interfeței grafice

Pentru crearea interfeței grafice, s-a utilizat JavaFX, menținând consistența în culori și formatare cu ajutorul unor clase ajutătoare: *Colors* (care salvează culorile personalizate folosite în proiect, cum ar fi roșu și verde) și *ComboBoxOptions* (deoarece un *ComboBox* nativ nu poate avea valori separate pentru textul afișat pe ecran și valoarea folosită în cerere, această clasă permite acest lucru).

Conform metodologiei JavaFX, view-ul este construit din două părți: fișierul .fxml și o clasă Controller care se ocupă de interacțiunea cu aplicația. În acest controller, sunt definite variabile pentru câmpurile de text pentru P și Q (polinoame), mesajul de eroare, ComboBox și input-ul utilizatorului. În ComboBox sunt enumerate operațiile posibile pe care le poate realiza utilizatorul. Atunci când se apasă pe câmpul unui polinom, acesta devine selectat, inputul primește valoarea acestuia și culoarea se schimbă în verde. Atunci când butonul de trimitere este apăsat, este creată o cerere și este trimisă către router (vezi secțiunea de Routing).

A screenshot of a computer

Description automatically generatedA white text box with black text

Description automatically generatedAceastă interfață este integrată în restul aplicației prin intermediul unui sistem de view-uri. Acest sistem are la bază o clasă *Window* de tip singleton care oferă un API pentru controllere, prin intermediul căruia pot fi setate view-urile după numele acestora. Pentru a putea transmite date de la controller la view, a fost creată o clasă Session care se comportă similar cu cea implementată de browser.

### Structura pachetelor

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedPachetele pentru această aplicație au fost structurate într-o manieră care permite modularizarea aplicației, făcând ușor convertirea modului de lucru într-un framework. Pachetele care se ocupă de gestionarea flow-ului de date sunt separate de pachetul în care se poate găsește logica afacerii și implementarea acesteia (*app*). În *resources* se pot găsi fișiere folosite de JavaFX views, CSS și imagini.

**Implementare**

Pentru implementare am ales un sistem bazat pe acțiuni. Pentru fiecare funcție pe care trebuie să o îndeplinească aplicația este creat un contract (o interfață) care descrie modul de funcționare. Pentru fiecare contract se pot găsi una sau mai multe implementări (în funcție de cerințele afacerii). Implementarea algoritmilor pentru operațiile matematice au fost discutați în *Capitolul 2: Proiectarea algoritmilor de prelucrare a polinoamelor*

### Containere de injecție de dependințe

Asocierea unui contract cu o implementare se realizează prin intermediul unui Container de injecție de dependințe (DI Container, găsit în aplicație sub numele de *AppProvider*). El ne permite să ne referim în cod direct la contract, fără să ne intereseze implementarea. Asocierea între contract și implementare se realizează în *Main*, la începutul aplicației. Codul următor demonstrează crearea legăturii:

AppProvider.*set*(*GeneratesRandomNumber*.*class*, *new* GenerateRandomNumber());  
AppProvider.*set*(*GeneratesNewClients*.*class*, *new* GenerateRandomClient());  
AppProvider.*set*(*GeneratesNewCashRegisters*.*class*, *new* GenerateNewCashRegister());

Modul de funcționare este următorul:

*public class* AppProvider {  
 *private final Map*<Class<?>, Object> bindings = *new* HashMap<>();  
 *private static final* AppProvider ***instance*** = *new* AppProvider();  
  
 *public static* <T> *void* set(Class<T> clazz, T target) {   
 ***instance***.bindings.put(clazz, target);   
 }  
  
 *public static* <T> T get(Class<T> clazz) {   
 *return* clazz.cast(***instance***.bindings.get(clazz));   
 }

}

### Routing

După cum a fost discutat în secțiunea anterioară, obiectivul principal al arhitecturii este să creeze un mediu ușor de manipulat și extins. Deși a fost prezentat fluxul router-ului anterior, în această secțiune se dorește o expunere mai detaliată a modului în care rutele sunt integrate în aplicație. Codul de mai jos oferă o ilustrare comprehensivă a procesului de definire a rutelor. Fiecare rută este compusă dintr-un nume unic, utilizat pentru navigare, precum și un controller asociat și o acțiune specifică, localizată în interiorul controller-ului respectiv. Această abordare nu numai că simplifică gestionarea rutelor în aplicație, dar și permite o mai mare flexibilitate în adaptarea și extinderea funcționalității acesteia în viitor.

Router.*add*(*new* Route("start", StartController.*class*, "index"));  
Router.*add*(*new* Route("loading", LoadingController.*class*, "index"));  
Router.*add*(*new* Route("simulation", SimulationController.*class*, "index"));  
Router.*add*(*new* Route("logs", LogController.*class*, "index"));  
Router.*add*(*new* Route("simulation/create", SimulationController.*class*, "store"));

### Generarea simulării

Pentru simulare am creat un model **Simulation** care are rol de a seta parametri și dependințele simulării. Prin intermediul metodei *SetState()* o nouă simulare poate începe.

*public static void* setState(

Integer clients, Integer queues, Integer simulationTime, Integer arrivalTimeMin, Integer arrivalTimeMax, Integer serviceTimeMin, Integer serviceTimeMax

) {  
 instance.id = System.*currentTimeMillis*();  
 instance.logger = *new* Logger();  
 instance.statisticsManager = *new* StatisticsManager();  
 instance.queues = queues;  
 instance.clientsToGenerate = clients;  
 instance.simulationTime = simulationTime;  
 instance.arrivalTimeMin = arrivalTimeMin;  
 instance.arrivalTimeMax = arrivalTimeMax;  
 instance.serviceTimeMin = serviceTimeMin;  
 instance.serviceTimeMax = serviceTimeMax;  
 Client.*resetId*();  
}

Apoi simularea poate începe. Întregul proces al simulării rulează asincron pentru a nu afecta performanța aplicației. Primul pas este crearea numărului dorit de case de marcat, urmat de rularea efectivă a simulării.

*public static void* start() {  
 *GeneratesNewCashRegisters* registerGenerator = AppProvider.*get*(*GeneratesNewCashRegisters*.*class*);  
  
 CompletableFuture.*supplyAsync*(() -> registerGenerator.handle(instance.queues))  
 .thenAcceptAsync((ArrayList<CashRegister> registers) -> *new* SimulationRunner(  
 instance.clientsToGenerate,  
 registers,  
 instance.simulationTime,  
 instance.serviceTimeMin,  
 instance.serviceTimeMax,  
 instance.arrivalTimeMax,  
 instance.arrivalTimeMin  
 ).run()  
 );  
}

Simularea rulează sincron toate intervalele de timp și asincron procesarea intervalului de timp. Prima dată este generat un număr aleator de clienți, apoi sunt împărțiți la casele de marcat. Acest proces este urmat de procesarea caselor de marcat, fiecare casă de marcat este procesată pe un alt thread.

*public void* run() {  
 *for* (*int* i = 0; i <= simulationTime; i++) {  
 *int* time = i;  
  
 *int* clientsArrivedNumber = getClientsArrivedNumber(time, AppProvider.*get*(*GeneratesRandomNumber*.*class*));  
 clientsToGenerate -= clientsArrivedNumber;  
  
 *List*<Client> waitingClients = Collections.*synchronizedList*(*new* ArrayList<>());  
 HashMap<Integer, Pair<Integer, Client>> registersLogs = *new* HashMap<>();  
  
 *try* {  
 CompletableFuture  
 .*supplyAsync*(() -> AppProvider.*get*(*GeneratesNewClients*.*class*).handle(clientsArrivedNumber, time, serviceTimeMin, serviceTimeMax))  
 .thenAcceptAsync(clients -> {  
 dispatchClients(clients, registers);  
 Simulation.*getStatisticManager*().addNewClients(clients).computeAverageServiceTime().computeAverageWaitingTime();  
  
 *List*<CompletableFuture<Void>> futures = registers.stream()  
 .map(register -> CompletableFuture.*runAsync*(() -> {  
 register.processClient();  
 registerLog(register, registersLogs, waitingClients);  
 }))  
 .toList();  
  
 CompletableFuture.*allOf*(futures.toArray(*new* CompletableFuture[0])).join();  
  
 Simulation.*addLog*(time, waitingClients, registersLogs);  
 Simulation.*setLatestProcessedTime*(time);  
 }).get();  
 } *catch* (InterruptedException | ExecutionException e) {  
 *throw new* RuntimeException(e);  
 }  
 }  
}

Se poate observa cum log-uri și statisticile sunt generate îl alte clase.

### Interfața grafică

În procesul de afișare al unui view pot apărea trei tipuri de erori potențiale. Pentru fiecare dintre aceste situații, am stabilit un set de coduri de eroare, menite să faciliteze procesul de depanare al programului. Conform convenției stabilite, codul de eroare este alcătuit din codul procesului care a generat eroarea, codul acțiunii care a cauzat-o și pasul la care a intervenit aceasta. Deoarece doarclasa Window are capacitatea de a genera astfel de erori și este definită ca primul element, aceasta va fi identificată cu codul 1. Mai jos găsiți lista completă a erorilor definite:

* Fereastra nu a fost creată: cod de stare 110
* View-ul nu a fost creat: cod de stare 120
* Imposibilitate de citire din fișier: cod de stare 121

# Concluzii

În urma dezvoltării proiectului calculatordepolinoame.ro, am acumulat cunoștințe substanțiale și am identificat perspective promițătoare pentru evoluția ulterioară a aplicației.

În ceea ce privește noțiunile dobândite, am ajuns la o înțelegere profundă a mecanismelor Dependency Injection (DI) și a rolului fundamental pe care DI Containerul îl joacă în gestionarea dependențelor în cadrul unei aplicații. De asemenea, am dezvoltat o perspectivă mai nuanțată asupra framework-urilor de backend, cu accent pe Laravel, și am înțeles modul în care acestea facilitează dezvoltarea aplicațiilor web complexe și scalabile.

Privind către viitor, am identificat mai multe direcții de interes pentru dezvoltarea ulterioară a aplicației. Printre acestea se numără extinderea capacității de gestionare și raportare a erorilor, optimizarea performanței prin implementarea unui sistem eficient de caching pentru operațiile frecvent utilizate, dezvoltarea unei funcționalități de istoric pentru monitorizarea și analiza operațiilor anterioare și extinderea funcționalității polinoamelor pentru a permite utilizatorilor să opereze cu funcții în calitate de coeficienți și exponente pentru monoame.

# Bibliografie

1. Compleatable futures: <https://www.baeldung.com/java-completablefuture>